




C++ Strings

FRIDAY, APRIL 4

In today's lecture, we explore operations on the C++ `string` datatype.

-  Readings: [Text](#) 3.1-3.7
-  [Lecture quiz on Canvas](#)
-  [Lecture video on Canvas](#)

Lecture Video

Click to sign in and play video

Contents

1. A Note About Lecture Note Notations
2. Library Functions and Exam Reference Sheets
3. Announcements
4. Lightning Review
5. Void Functions
6. Function Placement and Functional Prototypes
7. Passing Parameters and Returning Values from Functions
8. Pass-by-Value Functions
9. **Supplementary; Please Review** A Slight Diversion: Variable Scope
10. Pass-by-Reference Functions
11. Qt Creator's Indication of Pass-by-Reference
12. ASCII Characters (and Typecasting!)
13. Magic Numbers
14. The Nature of Strings in C++
15. C++ String Member Functions

16. Stanford C++ String Library (`strlib.h`)
17. Accessing (and Modifying) Individual Characters in Strings
18. Looping Through Strings
19. C++ Character Processing Library (`cctype`)
20. **Supplementary; Please Review** Character Processing: Additional Example
21. C++'s Two Types of Strings
22. Common Pitfall with Strings in C++
23. What's next?
24. Exam Prep

A Note About Lecture Note Notations

Throughout today's notes, there are a few paragraphs and examples labeled (*Not covered in class.*) Those are supplemental notes designed to enrich your understanding of the material. Reviewing them is strongly recommended.

There are also entire sections of notes that have a **Supplementary; Please Review** tag on their section titles. Those are items that are slightly more critical for you to review, which we did not have time to discuss in class today.

Library Functions and Exam Reference Sheets

We talked about a lot of library functions today for strings and characters. You do not have to memorize all of those for your exams! We'll actually provide you with a reference sheet that has all those library functions. You for sure want to familiarize yourself with them so you know what's what and can solve problems efficiently on your exams, but you don't have to worry about committing them to memory. 😊

Announcements

- A0 is due tonight at 11:59 PM.
- Q0 (Syllabus Quiz) is due tonight at 11:59 PM.
- Note that Quiz #0 (the syllabus quiz) and Assignment #0 are separate items. Assignment #0 will not be found -- or submitted -- on Canvas; it's found on the course website's [Assignment 0 page](#).
- Section sign-up is due Sunday at 5 PM. Note that assignments are not made on a first-come, first-served basis; we will build a schedule that works for everyone based on all the availability we've gathered as of 5 PM on Sunday.
- You have three quizzes due next Wednesday (Apr. 9) at 1 PM (30 mins prior to lecture). This is a batch deadline that covers this week's lecture quizzes. Please create a system to help you stay on top of deadlines throughout the quarter, and be mindful of the unconventional time for quiz deadlines.
- LaIR starts this Sunday. Please be sure to start assignments early so you can get help in LaIR early, before the lines get too long. Please also be sure to head to LaIR with reasonable expectations. (See my comments about that at the top of class today.)
- A1 unlocks today (shortly after class) and is due next Friday (Apr. 11) at 11:59 PM. Please consider pushing through most of it ASAP. There are a few pieces toward the end of it that rely on next Monday's lecture, and you don't want to wait until next Monday to start the whole thing.
- Section starts next week! You will receive a notification early next week about your section assignment. Once those are pushed out, you will be able to make adjustments in Paperless.

Lightning Review

We picked right back up today with a review exercise that involved fixing the following code:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    numCupcakes = 5

    if (numCupcakes == 1 || 2)
    {
        cout << "Uh oh! We're running low on cupcakes!" << endl
    }
}
```

Here's the corrected version:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int numCupcakes = 5;

    if (numCupcakes == 1 || numCupcakes == 2)
    {
        cout << "Uh oh! We're running low on cupcakes!" << endl;
    }

    return 0;
}
```

Void Functions

In this section, we define a few functions other than `main()` and explore return types, return values, and parameters.

If a function takes no parameters, we simply leave the parameter list within its parentheses blank.

If we write a function that does some work for us, but which we never expect to produce a meaningful value at the end of its execution, then it needn't return a value at all. In this case, we give the function a return type of `void`, like so:

```
#include <iostream>
#include "console.h"
using namespace std;

void greet()
{
    cout << "hello :)" << endl;
}

int main()
{
    greet();
    return 0;
}
```

A `void` function does not require a `return` statement. We return to the function that called it when we reach the end of the function's definition.

Optionally, if we wish to leave a `void` function before its final line, we can simply `return` (without specifying a return value), like so:

```
#include <iostream>
#include "console.h"
using namespace std;

void processCupcakes(int numCupcakes)
{
    if (numCupcakes < 0)
    {
        cout << "Invalid number of cupcakes." << endl;
        return;
    }

    // One can imagine doing something useful with numCupcakes here.
    // If we pass a negative value to this function, the following line
    // will not print.
    cout << "We reached the last line of the function." << endl;
}

int main()
{
    processCupcakes(-3);
    return 0;
}
```

Function Placement and Functional Prototypes

I mentioned again today that the C++ compiler processes your code line-by-line, starting at the very first line. A consequence of that is that if the compiler encounters a function call for a function that hasn't been defined yet (or for which we have not yet written the requisite `#include` statement), it goes kaput. For example:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    cout << square(5) << endl; // ERROR: square() function not yet defined.
    return 0;
}

int square(int x)
{
    return x * x;
}
```

What's happening above is that when we hit the `cout` line, C++ goes, "What's this `square()` function? Never heard of it." And it stops trying to compile the program in protest.

There are two ways around this:

1. Move the definition of the `square()` function above `main()` .
2. Introduce a functional prototype!

A **functional prototype** is just a function signature with a semicolon. A **function signature** is that first line of your function's definition that gives the return type, function name, and parameter list. Here's a functional prototype in action:

```
#include <iostream>
#include "console.h"

using namespace std;

// functional prototype
int square(int x);

int main()
{
    cout << square(5) << endl; // This is okay now.
    return 0;
}

int square(int x)
{
    return x * x;
}
```

This time, when C++ reaches the `cout` line, even though it doesn't have a definition for the `square()` function, it has a good idea (from having already seen the functional prototype) of how that function should be called,

including the number and type of arguments it takes and its return type. It checks out the `cout` line and says, "Alrighty. I don't know how the `square()` function works yet, but I do know that the way you're calling it here fits with how I expect it to be called. Good work." It then carries on and successfully compiles the program.

(*Not covered in class.*) Note that if we never defined the `square()` function, the program would not compile, even if we had a functional prototype. The compiler would fail at the very end of the file, when it realized there was no more code to process and it never got a definition for `square()` :

```
#include <iostream>
#include "console.h"

using namespace std;

// functional prototype
int square(int x);

int main()
{
    cout << square(5) << endl;
    return 0;
}

// ERROR: Reached end of code without ever defining square() function.
```

Passing Parameters and Returning Values from Functions

The following `square()` function is designed to take a single integer parameter, `x`, and return x^2 . To set this function up to accept a parameter, we give a full declaration for that variable within the parentheses in the function signature (the line with the function's name, return type, and parameter list).

```
#include <iostream>
#include "console.h"
using namespace std;

// We pass an integer to this function, and it returns an integer.
int square(int x)
{
    return x * x;
}

int main()
{
    square(5);
    return 0;
}
```

However, when we run the above program, nothing gets printed to the screen. That's because we never did anything in `main()` with the return value of our call to `square(5)`. We must capture that return value. Here are two ways to print the return value to the screen:

```
#include <iostream>
#include "console.h"
using namespace std;

// We pass an integer to this function, and it returns an integer.
int square(int x)
{
    return x * x;
}

int main()
{
    // Option 1: Send the return value of square(5) directly to cout.
    cout << square(5) << endl;

    // Option 2: Store the return value in a variable and print that.
    int result = square(5);
    cout << result << endl;

    return 0;
}
```

The program now produces the following output:

```
25
25
```

(*Not covered in class.*) If we want our function to print the result directly, we could write it as follows. Notice that the modified behavior of this function is reflected both in its modified name and the `void` return type:

```
#include <iostream>
#include "console.h"
using namespace std;

void printSquare(int x)
{
    cout << x * x << endl;
}

int main()
{
    printSquare(5);
    return 0;
}
```

Pass-by-Value Functions

We then explored this example of a **pass-by-value** function and saw that when we call `foo(n)`, that creates an additional `n` variable in memory that is separate from the `n` variable in `main()`:

pass-by-value approach:

```
#include <iostream>
#include "console.h"
using namespace std;

void foo(int n)
{
    n++;
}

int main()
{
    int n = 3;
    foo(n);

    // This prints 3. The ++ operator in foo() only increments foo()'s local copy
    // of n -- not our n variable back in main().
    cout << n << endl;

    return 0;
}
```

Here's what's happening in memory. Notice that `main()` and `foo()` have separate `n` variables:

```
main()
+-----+
|  n    |
| +-----+ |
| |  3  | |
| +-----+ |
+-----+

foo()
+-----+
|  n    |
| +-----+ |
| |  4  | |
| +-----+ |
+-----+
```

output:

3

Supplementary; Please Review A Slight Diversion: Variable Scope

Variables only exist within the code blocks where they are declared -- or, in the event that you declare a variable in the header of a for-loop, that variable only exists within the loop. We refer to the regions of code that can refer to a particular variable as the variable's **scope**.

We saw an example of that above when declaring an integer `n` in two separate functions. Here is another example:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    for (int i = 0; i < 5; i++)
    {
        int q = 5;
        cout << i << endl;
    }

    cout << "i is: " << i << endl; // ERROR: undeclared variable
    cout << "q is: " << q << endl; // ERROR: undeclared variable

    return 0;
}
```

In the code above, we say that the `i` and `q` variables only exist within the scope of the for-loop. To refer to them outside the for-loop, they would have to be declared elsewhere. For example:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int i;
    int q;

    for (i = 0; i < 5; i++)
    {
        q = 5;
        cout << i << endl;
    }

    cout << "i is: " << i << endl; // OKAY
    cout << "q is: " << q << endl; // OKAY

    return 0;
}
```

Pass-by-Reference Functions

We then saw an example of a **pass-by-reference** function in C++. Here, we have turned `foo()`'s `n` variable into a **reference** by appending an ampersand (`&`) to its data type.

In this example, `foo()`'s `n` is masquerading as a normal integer, but under the hood, it's actually acting as a portal to `n` back in `main()`. In this way, it **refers** to `main()`'s `n` variable (hence the term "reference"). When we execute the `n++` line, the program actually goes back to `main()` and modifies the value of `n` that we find there.

pass-by-reference approach:

```
#include <iostream>
#include "console.h"
using namespace std;

// This is now a pass-by-reference function. Note the addition of the ampersand.
void foo(int& n)
{
    n++;
}

int main()
{
    int n = 3;
    foo(n);

    // This prints 4. The ++ operator in foo() increments n back in main(), since
    // foo()'s copy of n is tied directly to this one.
    cout << n << endl;

    return 0;
}
```

Here's what's happening in memory. Notice that `foo()`'s `n` variable now acts as a wormhole to `main()`'s `n` variable:

```

main()
+-----+
|  n    |
| +-----+
| |  4  |
| +-----+
+-----+
      |
      |
foo()  |
+-----+-----+
|  n    |         |
| +-----+-----+
| |  @  |         |
| +-----+-----+
+-----+-----+

```

output:

4

There are at least two great reasons to use references in C++:

1. They help us get around the fact that functions in C++ only return a single value. If we want to write a function that produces more than one value that we're interested in capturing, references help us do that. Some examples include:
 - a. A `swap()` function that takes two integers by reference and results in them swapping values. This is a common operation in all kinds of algorithms -- especially sorting algorithms, which we'll explore later this quarter.
 - b. A `solveQuadratic()` function that takes three integers by value (`a` , `b` , and `c` , the coefficients for the quadratic equation) and two integers by reference (`result1` and `result2`) and uses those references to communicate the roots of the equation back to the calling function. See Section 2.5 of the course textbook for the implementation.
2. They can be used to save time and space when working with large amounts of data. For example, if we have a string that contains 1 GB of text, passing it to a function by value causes us to use another GB of memory to create a separate copy of it, and it takes time to copy that GB of text from one variable to another. In contrast, if we pass the string by reference, we use a tiny amount of memory (typically just 64 bits!) to establish the tether between that reference variable and the original variable being referenced. Because we use so little memory, the transaction involved is very fast, too.

The other example we saw involved a treasure hunt. A pass-by-value approach to raiding a series of treasure hoards makes no sense, as it does not empty out the original hoards in memory:

pass-by-value approach:


```
#include <iostream>
#include "console.h"
using namespace std;

int treasureHunt(int h1, int h2, int h3)
{
    int myTreasarrrrrr = 0;

    myTreasarrrrrr += h1;
    h1 = 0;

    myTreasarrrrrr += h2;
    h2 = 0;

    myTreasarrrrrr += h3;
    h3 = 0;

    return myTreasarrrrrr;
}

int main()
{
    int hoard1 = 350;
    int hoard2 = 43;
    int hoard3 = 5003;

    int totalBooty = treasureHunt(hoard1, hoard2, hoard3);

    cout << "Total Booty: " << totalBooty << endl << endl;

    // We've pirated and pillaged, but these hoards don't reflect that.
    cout << "hoard1: " << hoard1 << endl;
    cout << "hoard2: " << hoard2 << endl;
    cout << "hoard3: " << hoard3 << endl;

    return 0;
}
```

output:

```
Total Booty: 5396

hoard1: 350
hoard2: 43
hoard3: 5003
```

A minor adjustment turns this into a pass-by-reference function and gives us the desired behavior of wiping out the treasure hoards as we pillage:

pass-by-reference approach:

```

#include <iostream>
#include "console.h"
using namespace std;

int treasureHunt(int& h1, int& h2, int& h3)
{
    int myTreasarrrrrr = 0;

    myTreasarrrrrr += h1;
    h1 = 0;

    myTreasarrrrrr += h2;
    h2 = 0;

    myTreasarrrrrr += h3;
    h3 = 0;

    return myTreasarrrrrr;
}

int main()
{
    int hoard1 = 350;
    int hoard2 = 43;
    int hoard3 = 5003;

    int totalBooty = treasureHunt(hoard1, hoard2, hoard3);

    cout << "Total Booty: " << totalBooty << endl << endl;

    // These are now zeroed out.
    cout << "hoard1: " << hoard1 << endl;
    cout << "hoard2: " << hoard2 << endl;
    cout << "hoard3: " << hoard3 << endl;

    return 0;
}

```

output:

```

Total Booty: 5396

hoard1: 0
hoard2: 0
hoard3: 0

```

Qt Creator's Indication of Pass-by-Reference

Note that when we call a function, Qt Creator uses italics to indicate whether any parameters are pass-by-value. Unitalicized parameters are pass-by-value. (See calls to `treasureHunt()` above.)

ASCII Characters (and Typecasting!)

We then explored the ASCII values underlying the `char` data type in C++. ASCII is an international standard for character representation. To force C++ to show us the ASCII values underlying specific characters, I used C++'s "function-style" typecast, which involved placing a `char` variable in the parentheses of `int(...)` to cajole C++ into treating it as an `int` within that context.

Furthermore, we saw that because chars in C++ are actually just ints behind the scenes, we can apply comparison and arithmetic operators to them. For example:

```

#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    for (char ch = 'a'; ch <= 'z'; ch++)
    {
        // typecast!
        cout << ch << " (" << int(ch) << ")" << endl;
    }

    return 0;
}

```

output:

```
a (97)
b (98)
c (99)
d (100)
e (101)
f (102)
g (103)
h (104)
i (105)
j (106)
k (107)
l (108)
m (109)
n (110)
o (111)
p (112)
q (113)
r (114)
s (115)
t (116)
u (117)
v (118)
w (119)
x (120)
y (121)
z (122)
```

Magic Numbers

I then presented the challenge of printing ordinal numbers 1 through 26 next to those characters. A simple tweak is as follows (see highlighted code):

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    for (char ch = 'a'; ch <= 'z'; ch++)
    {
        cout << ch << " (" << int(ch) - 96 << ")" << endl;
    }

    return 0;
}
```

output:

```
a (1)
b (2)
c (3)
d (4)
e (5)
f (6)
g (7)
h (8)
i (9)
j (10)
k (11)
l (12)
m (13)
n (14)
o (15)
p (16)
q (17)
r (18)
s (19)
t (20)
u (21)
v (22)
w (23)
x (24)
y (25)
z (26)
```

A problem with this approach is that we're hard-coding an integer that to the average reader might seem so mysterious and opaque that they would look at the code and say, "What does that number mean? How does that even work? It looks like *magic*!"

We call such literal values "**magic numbers**," and it's considered best practice to avoid them. You might be able to

get away with a really obvious literal in your code (such as the number 26 when working with the alphabet), but the 96 is rather obscure. We rewrote the code as follows, replacing the magic number with `('a' - 1)`:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    for (char ch = 'a'; ch <= 'z'; ch++)
    {
        cout << ch << " (" << int(ch) - ('a' - 1) << ")" << endl;
    }

    return 0;
}
```

The Nature of Strings in C++

We then dove into the nature of strings in C++ and saw two key points on that topic:

1. Strings are arrays of characters.

At its core, a string is really just an array of characters. That is, a string is a block of characters that are contiguous in memory that are indexed from 0 through $(n - 1)$, where n is the overall number of characters in the string. For example, the string "hello there" is represented as follows:

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| 'h' | 'e' | 'l' | 'l' | 'o' | ' ' | 't' | 'h' | 'e' | 'r' | 'e' |
+-----+-----+-----+-----+-----+-----+-----+-----+
  0     1     2     3     4     5     6     7     8     9    10
```

(Important Note) Recall that strings in C++ must be given in double quotes, not single quotes. We only use single quotes for single characters.

2. Strings are objects (not primitive data types).

I also pointed out that when we declare a string variable in the Qt Creator, the default color for the word `string` is different from the color used for `int`, `float`, `double`, and `char`. That's because strings in C++ are fundamentally different from the latter data types, which are called **primitive types**. Primitive types form the foundation for all other data types in C++.

A string, in contrast, is an **object**. We will explore objects in more detail closer to the mid-term. For now, I'd like you to know that this means a string is a bundle -- not just of the character array representing that string, but also of a bunch of built-in functions called **member functions** that we can access by using the dot operator (`.`) on one of our strings.

For example:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    string s = "hello there";

    // Notice that we are not passing s to a length function. Rather, we use the dot
    // operator on the s variable to call a length() function that is packed up with
    // the string and which gives us the length of that string (11, as there are 11
    // characters in "hello there"). (The space counts a character.)

    cout << s.length() << endl;

    return 0;
}
```

output:

```
11
```

C++ String Member Functions

String objects in C++ have a ton of built-in functions. To use these functions, we technically have to `#include <string>` . However, including either `<iostream>` or `"console.h"` actually indirectly causes the string library to be imported behind the scenes, so we don't need to do that directly in order for our code to compile and run.

Here's an index of some of the most handy functions we can call using a string's dot operator in C++:

Member Function	Description
<code>s.append(str)</code>	add text to the end of a string
<code>s.compare(str)</code>	return -1, 0, or 1 depending on relative ordering
<code>s.erase(index, length)</code>	delete text from a string starting at given index
<code>s.find(str)</code> <code>s.rfind(str)</code>	return first or last index where the start of <code>str</code> appears in this string (returns <code>string::npos</code> if not found)
<code>s.insert(index, str)</code>	add text into string at a given index
<code>s.length()</code> <code>s.size()</code>	return number of characters in the string
<code>s.replace(index, len, str)</code>	replace <code>len</code> chars at given index with new text
<code>s.substr(start, length)</code> <code>s.subsr(start)</code>	return a new string with the next <code>length</code> chars beginning at <code>start</code> (inclusive); if <code>length</code> is omitted, grabs till end of string

In class, we saw an example in class of the `substr()` function in action:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    string s = "hello there";

    // Start at index 1 in the string and pull out the first 4 characters there.
    cout << s.substr(1, 4) << endl;

    return 0;
}
```

output:

```
ello
```

Stanford C++ String Library (`strlib.h`)

Similarly, the Stanford C++ string library has some powerful functions for processing strings. This library is really well documented and fairly straightforward.

Function Name	Description
<code>endsWith(str, suffix)</code> <code>startsWith(str, prefix)</code>	returns <code>true</code> if the given string begins or ends with the given prefix/suffix text
<code>integerToString(int)</code> <code>realToString(double)</code> <code>stringToInteger(str)</code> <code>stringToReal(str)</code>	returns a conversion between numbers and strings
<code>equalsIgnoreCase(s1, s2)</code>	returns true if <code>s1</code> and <code>s2</code> have same chars, ignoring case
<code>toLowerCase(str)</code> <code>toUpperCase(str)</code>	returns an upper/lowercase version of a string (pass-by-value!)
<code>trim(str)</code>	returns string with surrounding whitespace removed

(*Super Handy Productivity Tip*) If you `#include "strlib.h"` in a Qt Creator project, you can CTRL+CLICK (CMD+CLICK on Mac) on `strlib.h` to go into that file and see a list of functional prototypes with descriptions of

what those functions do. Alternatively, you can click over to the Resources page on the course website, click "Stanford Library documentation," then click [strlib.h](https://en.cppreference.com/strlib) on that page.

We saw the following example of one of those functions being called:

```
#include <iostream>
#include "console.h"
#include "strlib.h"
using namespace std;

int main()
{
    string s = "hello there";
    toUpperCase(s);
    cout << s << endl;

    return 0;
}
```

output:

```
hello there
```

Notice that this does *not* convert `s` to uppercase! That's because `toUpperCase()` is a pass-by-value function. We can determine that in the Qt Creator simply by observing that when we call `toUpperCase(s)`, the `s` parameter does not get italicized.

`toUpperCase(s)` returns the uppercase version of the string it receives. To get this example working, we need to store that return value somewhere. Here's the working version of this example:

```
#include <iostream>
#include "console.h"
#include "strlib.h"
using namespace std;

int main()
{
    string s = "hello there";
    s = toUpperCase(s);
    cout << s << endl;

    return 0;
}
```

output:

```
HELLO THERE
```

Accessing (and Modifying) Individual Characters in Strings

We saw that we can access (and change!) individual characters in C++ strings. This is unusual compared to Python and Java, where we can't just tweak one character. If we want to modify a Python or Java string, we need to call functions or apply operators that ultimately build new strings in the background.

We say C++'s strings are **mutable** (able to be changed), whereas Python and Java strings are **immutable**. If a string appears to be changing in Python or Java, it's really not; what you're seeing is really a new, separate string.

Along those lines, we examined the following code:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    string s = "hello";
    cout << s << endl;

    // This changes the character at index 0. I read "s[0]" out loud as "s-sub-zero."
    s[0] = 'Y';
    cout << s << endl;

    // We can concatenate a single character to a string using the + operator. We know
    // the w below is just a character (not a string) because it's in single quotes.
    s += 'w';
    cout << s << endl;

    // We can also concatenate an entire string.
    s += " squashes";
    cout << s << endl;

    return 0;
}
```

output:

```
hello
Yello
Yellow
Yellow squashes
```

Looping Through Strings

We then reviewed two ways to loop through strings:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    string s = "hello";

    // We can loop through a string using a for-loop.
    for (int i = 0; i < s.length(); i++)
    {
        cout << i << ": " << s[i] << endl;
    }

    // We can also use a for-each loop (aka "range-based loop"), but this approach
    // doesn't give us a handy variable to tell us what index we're at in the string.
    cout << endl;
    for (char ch : s)
    {
        cout << ch << endl;
    }

    return 0;
}
```

output:

```
0: h
1: e
2: l
3: l
4: o

h
e
l
l
o
```

C++ Character Processing Library (`cctype`)

We talked briefly about the `cctype` library is chock full of char processing functions, almost all of which take a

single char argument and are named `isXXXXX()` (where `XXXXX` is some five-letter string indicating what type of character the function is checking for).

For your reference, here are the functions from that library. Please be sure to look over these on your own time so you have an idea of what functions are available to you as you start working on your programming assignments for this course:

Member Function	Description
<code>isalnum(ch)</code>	checks if a character is alphanumeric
<code>isalpha(ch)</code>	checks if a character is alphabetic
<code>islower(ch)</code>	checks if a character is a lowercase alphabetic character
<code>isupper(ch)</code>	checks if a character is an uppercase alphabetic character
<code>isdigit(ch)</code>	checks if a character is a digit
<code>isxdigit(ch)</code>	checks if a character is a hexadecimal character
<code>iscntrl(ch)</code>	checks if a character is a control character
<code>isgraph(ch)</code>	checks if a character is a graphical (i.e., visible) character
<code>isspace(ch)</code>	checks if a character is a space character (typically tab or space)
<code>isblank(ch)</code>	checks if a character is a blank character
<code>isprint(ch)</code>	checks if a character is a printing character according to locale
<code>ispunct(ch)</code>	checks if a character is punctuation (visible non-alnum/non-space)
<code>toupper(ch)</code>	converts a character to uppercase (pass-by-value!)
<code>tolower(ch)</code>	converts a character to lowercase (pass-by-value!)

In class, we saw an example similar to the following that illustrated the use of `isupper()`. This example is a bit more elaborate and instead focuses on `isalpha()`, which tells us whether a character is alphabetical or not:

```
#include <cctype>
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    string s1 = "yellow";
    string s2 = "$assy";

    // Should print "yes" because s1[0] ('y') is an alphabetic character.
    if (isalpha(s1[0]))
    {
        cout << "yes" << endl;
    }
    else
    {
        cout << "no" << endl;
    }

    // Should print "no" because s2[0] ('$') is not an alphabetic character.
    if (isalpha(s2[0]))
    {
        cout << "yes" << endl;
    }
    else
    {
        cout << "no" << endl;
    }

    return 0;
}
```

Supplementary; Please Review Character Processing: Additional Example

Here is an additional example of a character processing function for you to review. It prints all the lowercase

alphabetic characters from a string. This is more sophisticated than some of the chunks of code we have written so far in class. Can you see how this is working?

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    string password = "p4ssw0rd";
    string alphaPortion;

    for (int i = 0; i < password.length(); i++)
    {
        // checks whether password[i] is a lowercase char
        if (password[i] >= 'a' && password[i] <= 'z')
        {
            alphaPortion += password[i];
        }
    }

    cout << alphaPortion;

    return 0;
}
```

output:

psswrđ

(*Important Note*) (*Not covered in class.*) We have talked a bit about the fact that uninitialized variables are problematic in C++. That doesn't apply to C++'s string variables, though. Those actually get auto-initialized to an empty string (double quotes with nothing -- not even a space -- between them: "") if you don't give them some other value. So, the use of the uninitialized `alphaPortion` variable above is not problematic.

A variation on this approach relies on the `isalpha()` function from C++'s `cctype` library:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    string password = "p4ssw0rd";
    string alphaPortion;

    for (int i = 0; i < password.length(); i++)
    {
        if (isalpha(password[i]))
        {
            alphaPortion += password[i];
        }
    }

    cout << alphaPortion << endl;

    return 0;
}
```

output:

psswrđ

C++'s Two Types of Strings

I mentioned this briefly toward the end of class today -- not because it's super important for you to have a deep understanding of this, but because I want you to have a reference on this topic in case you run into problems with this while working with strings in your programming assignments this quarter:

C++ actually has two types of strings. We have seen C++'s `string` variable type, but any time we type a **string literal** (i.e., a hard-coded string in double quotes), we are actually getting a C-style string.

Applying operators to two C-style strings is dangerous and can lead to code that won't compile -- or worse: wonky, unexpected results.

However, we can apply operators to two strings if **at least** one of them is a C++ style string, and we can always get a C++ style string by assigning a C-style string to a C++ string variable.

Here are some examples to clarify:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    // BAD! Before we do the "=" operation, we resolve the right-hand side of that
    // operator. That involves trying to apply the "+" operator to two string literals,
    // which are C-style strings. C++ will not even allow this to compile.
    string s1 = "abc" + "xyz";

    // OKAY. We can assign C-style strings directly to C++ string variables.
    string s2 = "abc";
    string s3 = "xyz";

    // OKAY. Both s2 and s3 are C++ style strings. We can apply the "+" operator
    // to two C++ strings to concatenate them.
    string s4 = s2 + s3;
    cout << s4 << endl;

    // OKAY. This works, too! The right-hand side of the "=" operator gets resolved
    // first, and since s5 + "hello" involves at least one C++ style string, the
    // operation is good to go.
    string s5 = "hello";
    string s6 = s5 + " there";
    cout << s6 << endl;

    // OKAY. Same as above, but with (C + C++) instead of (C++ + C).
    string s7 = " there";
    string s8 = "hello" + s7;
    cout << s8 << endl;

    // OKAY. There is a function called string() that we can use to create a C++
    // style string out of a C-style string. As long as we apply it to at least one of
    // the C-style strings below, we end up with at least one C++ style string, and we
    // are then in the clear to use the concatenation operator.
    string s9 = "hello" + string(" there");
    cout << s9 << endl;

    // OKAY. Similarly, C++ allows us to typecast a C-style string to a C++ string.
    string s10 = "hello" + (string)" there";
    cout << s10 << endl;

    return 0;
}
```

Common Pitfall with Strings in C++

(*Not covered in class.*) C++ lets you access invalid indices when playing with strings. This can cause all kinds of wonky things to happen in your programs. For example:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    string s = "hello";

    // UH OH! Accessing invalid index! Who knows what will happen?
    cout << s[10] << endl;

    return 0;
}
```

output (which may be different when you run this on your system):



Going far enough out of bounds in your string can actually cause a program to crash:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    string s = "hello";

    // UH OH! Accessing invalid index! Who knows what will happen?
    cout << s[1000000000] << endl;

    return 0;
}
```

output (which may be different when you run this on your system):

```
*** The Day03Lecture program has terminated unexpectedly (crashed)
*** A segmentation fault (SIGSEGV) occurred during program execution

This error indicates your program attempted to dereference a pointer
to an invalid memory address (possibly out of bounds, deallocated, nullptr, ...)

*** To get more information about a program crash,
*** run your program again under the debugger.
```

What's next?

On Monday, we'll talk briefly about testing (including the `SimpleTest` framework in the Stanford C++ Libraries) and then delve into two abstract data types that are implemented in the Stanford libraries: `Vector` and `Grid` .

Exam Prep

Your biggest training grounds for the material we're covering in lecture will be (a) your weekly section meeting, where you'll get more detailed explanations of some of these topics as well as hands-on experience with practical applications, and (b) the assignments, which are constructed to guide you through the material and help you ramp up your understanding quite quickly.

Nonetheless, I will occasionally post additional practice problems for those who are looking for even more reinforcement and hands-on experience or stepping stones to help work up to the week's assignment and/or section problems.

1. (*Pass-by-Reference Functions*) Write a pass-by-reference function called `mySwap()` that takes two integers and swaps their values. Your function should pass the test case provided below. What should the return type be? What types should we use for the function parameters? (Note: The function is called `mySwap()` because there is already a native `swap()` function in C++.)

```
#include <iostream>
#include "console.h"
#include "SimpleTest.h"
using namespace std;

PROVIDED_TEST("basic test of swap() function with two integers")
{
    int a = 53;
    int b = 42;

    mySwap(a, b);

    EXPECT_EQUAL(a, 42);
    EXPECT_EQUAL(b, 53);
}

int main()
{
    runSimpleTests(ALL_TESTS);
    return 0;
}
```

Highlight for solution:

```

#include <iostream>
#include "console.h"
#include "SimpleTest.h"
using namespace std;

// Note: Alternatively, we could put this below main() and
// drop a functional prototype here.
void mySwap(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}

PROVIDED_TEST("basic test of swap() function with two integers")
{
    int a = 53;
    int b = 42;

    mySwap(a, b);

    EXPECT_EQUAL(a, 42);
    EXPECT_EQUAL(b, 53);
}

int main()
{
    runSimpleTests(ALL_TESTS);
    return 0;
}

```

2. (*Pass-by-Reference Functions*) Now that you have some practice with pass-by-reference functions from Exercise 1, see if you can close out the the notes and recreate the functionality of today's `treasureHunt()` function from scratch, without peeking back at the notes for assistance.

3. (*Pass-by-Reference Functions*) Trace through the following program by hand to see if you can determine the output it will produce. Then run compile and run the program to check your result.

```

#include <iostream>
#include "console.h"
using namespace std;

void mystery(int& b, int c, int& a)
{
    a++;
    b--;
    c += a;
}

int main()
{
    int a = 5;
    int b = 2;
    int c = 8;

    mystery(c, a, b);
    cout << a << " " << b << " " << c << endl;

    return 0;
}

```

4. Write a function that takes a string as its only parameter and returns the sum of all the ASCII values of the characters in that string. For example, `asciiSum("cat")` should return `'c' + 'a' + 't' = 99 + 97 + 116 = 312`. As part of this exercise, come up with the full function signature for `asciiSum()`.

5. Be sure to carefully review the notes labeled **Supplementary; Please Review**. There is an interesting character processing function you should look into, a brief note about how variable scope relates to loops, and a note about how the Qt Creator signals to users when they're passing variables to a function by reference.

6. Be sure to glance through the notes labeled (*Not covered in class.*) to enrich and/or solidify your understanding of some of the finer points of this material. Those extra nuggets of information might prove useful at some point this quarter.

Looking for more practice problems? Be sure to consult the course reader linked at the top of today's notes.

All course materials © Stanford University 2024. This content is protected and may not be shared, uploaded, or distributed.

Website programming by Julie Zelenski with modifications by Sean Szumlanski • Styles adapted from Chris Piech • This page last updated 2025-Apr-04